3806ICT - Assignment 1

Nick van der Merwe - s5151332 - nick.vandermerwe@griffithuni.edu.au

May 3, 2021

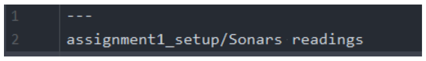
1 Introduction

In this piece a controller is defined for a robot using a proportional integral derivative controller. This is split into two tasks, defining the PID algorithm and the Kalman filter in the assignment. However, the architecture of the solution will contain four parts: the sonar wrapper, the PID service, the Kalman filter service and the controller. Essentially, the sonar wrapper will turn the sonar topic into a service so that the controller does not subscribe to it. The PID and the Kalman filter will be purely functions without any storage so that they can be reused in the future. Lastly, the controller node will manage all of the data.

This report is organised based on the content in the nodes. Each section will introduce the necessary theory and explain how that is converted into code. With that in mind, Section 2 contains the sonar wrapper, Section 3 covers the PID equation (not defining the constants), Section 4 runs over the kalman filter (not finding the variance), and Section 5 defines the controller. The controller contains how all the previous nodes are tied together, including how the constants in PID are defined and variance is found for the Kalman filter.

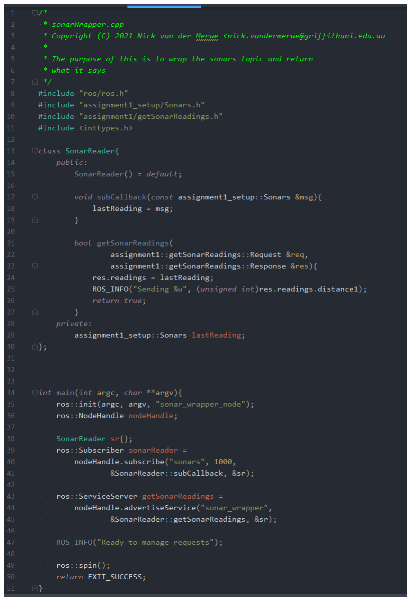
2 Sonar Wrapper Node

This is quite straight forwards as it only requires us to save the last published value in a variable. As a design decision to avoid global variables it was made as a class. The code is visible in Figure 1a and 1b.



```
---
assignment1_setup/Sonars readings
```

(a)



```cpp
/*
 * sonarWrapper.cpp
 * Copyright (C) 2021 Nick van der Merwe <nick.vandermerwe@griffithuni.edu.au
 *
 * The purpose of this is to wrap the sonars topic and return
 * what it says
 */
#include "ros/ros.h"
#include "assignment1_setup/Sonars.h"
#include "assignment1/getSonarReadings.h"
#include <inttypes.h>

class SonarReader{
    public:
        SonarReader() = default;

        void subCallback(const assignment1_setup::Sonars &msg){
            lastReading = msg;
        }

        bool getSonarReadings(
                assignment1::getSonarReadings::Request &req,
                assignment1::getSonarReadings::Response &res){
            res.readings = lastReading;
            ROS_INFO("Sending %u", (unsigned int)res.readings.distance1);
            return true;
        }
    private:
        assignment1_setup::Sonars lastReading;
};



int main(int argc, char **argv){
    ros::init(argc, argv, "sonar_wrapper_node");
    ros::NodeHandle nodeHandle;

    SonarReader sr{};
    ros::Subscriber sonarReader =
        nodeHandle.subscribe("sonars", 1000,
                &SonarReader::subCallback, &sr);

    ros::ServiceServer getSonarReadings =
        nodeHandle.advertiseService("sonar_wrapper",
                &SonarReader::getSonarReadings, &sr);

    ROS_INFO("Ready to manage requests");

    ros::spin();
    return EXIT_SUCCESS;
}
```

(b)

Figure 1: Sonar wrapper srv in a and code in b

3 Task 1: PID Node

Before the coding section, the mathematics behind what a proportional integral derivative controller does must be explained. In short, the goal of the controller is to minimise the error that is reported. As stated in the task, the formula to be used here is in Equation 1.

$$\begin{aligned} Y(t) &= K_p e(t) + K_d \frac{de(t)}{dt} + K_i \int_0^t e(t')dt' \\ &= P(t) + D(t) + I(t) \end{aligned} \tag{1}$$

Where $e(t)$ is the input error: in our case the distance from the bowl. t is the time, K_p is proportional gain, K_d is the derivative gain and K_i is the integral gain.

To make elaborating how this equation functions simpler, the first thing to do is to consider each of the terms separately as they each represent a part of **P**roportional **I**ntegral **D**erivative. To summarise, the proportional section represents the current error, the integral is the error in the past and the derivative is a prediction of the error into the future [M., 2000]. The logic behind this is that for the P term its just the current error, the integral is the area under the curve in the past and the derivative is the change at that specific point. These are each adjusted to the environment by using the K terms on each term.

Now the next issue is that this representation uses continuous time whereas in computational mathematics everything must be discrete - in this case our sonar only reports every 10ms. The exact maths for converting this into discrete values was covered in the lab content so it will not be copied over here, but rather explained.

The first step is to define time as discrete by making $T = 10ms$ and $t_i = T + t_i - 1$. Technically this would be all that's necessary, but the exact way to calculate the derivative and integral is not known. For these the conceptual meanings can be used: a derivative is the change in the output of a function in the smallest unit of time and an integral is the area under the curve from one time to another. Another fact is that technically a derivative forwards (current minus future) is the same as backwards (current minus last value) in continuous time due to it being infinitely small.

To take this and define them discretely with logic is simple. A derivative is its current value subtracted by its last value, all divided by T (since that is the smallest unit of time we have). Next the integral is simply the sum of all the previous error values times T. This can conceptually be seen as how the area of a square is found at a low level without multiplication: split it into strips of size 1 (in our case T) and add these areas together. Including the K terms, our equations are in Equations 2, 3 and 4.

$$P(t) = K_p e(t) \tag{2}$$

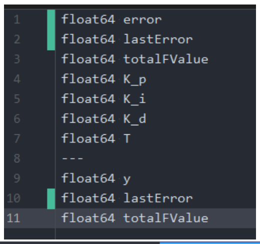
$$D(t) = K_d \frac{e(t_i) - e(t_{i-1})}{T} \tag{3}$$

$$I(t_i) = K_i \sum_{i=1}^{i} T e(t_i) \tag{4}$$

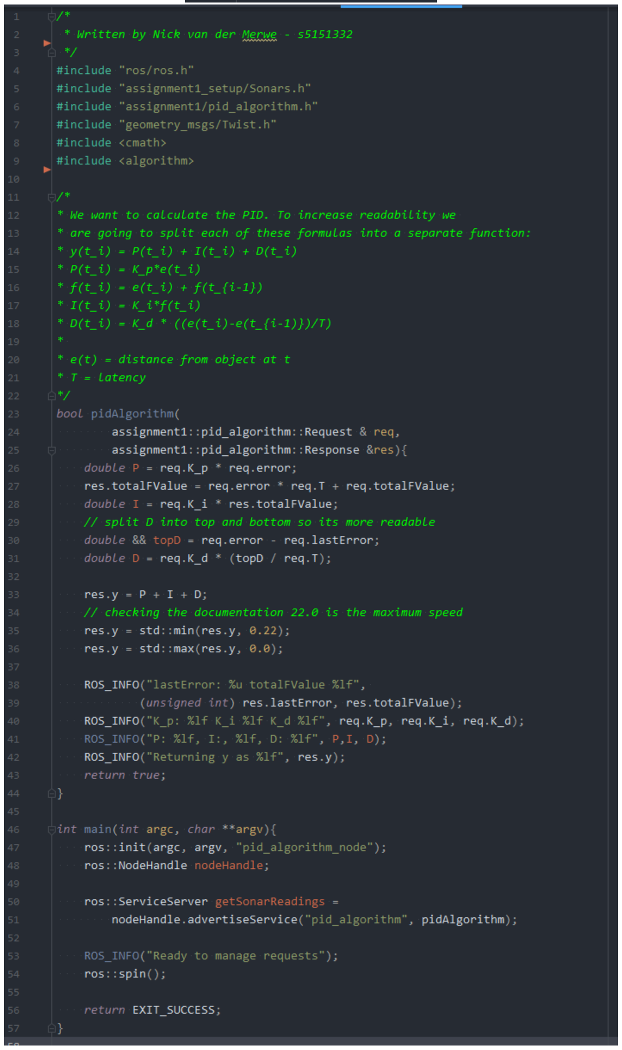
Programmatically, it would be inefficient to do the integral by summing up everything in this manner every time it is calculated, so realistically the value is simply returned on its own in the function so that it can just be added to once in the next run. This is represented as Equation 5.

$$\begin{aligned} I(t_i) &= K_i f(t) \\ f(t) &= e(t_i)T + f(t_{i-1}) \end{aligned} \tag{5}$$

The PID srv file and C++ code is visible in figure 2.



```
float64 error
float64 lastError
float64 totalFValue
float64 K_p
float64 K_i
float64 K_d
float64 T
---
float64 y
float64 lastError
float64 totalFValue
```



```cpp
/*
 * Written by Nick van der Merwe - s5151332
 */
#include "ros/ros.h"
#include "assignment1_setup/Sonars.h"
#include "assignment1/pid_algorithm.h"
#include "geometry_msgs/Twist.h"
#include <cmath>
#include <algorithm>

/*
* We want to calculate the PID. To increase readability we
* are going to split each of these formulas into a separate function:
* y(t_i) = P(t_i) + I(t_i) + D(t_i)
* P(t_i) = K_p*e(t_i)
* f(t_i) = e(t_i) + f(t_{i-1})
* I(t_i) = K_i*f(t_i)
* D(t_i) = K_d * ((e(t_i)-e(t_{i-1})})/T)
*
* e(t) = distance from object at t
* T = latency
*/
bool pidAlgorithm(
        assignment1::pid_algorithm::Request & req,
        assignment1::pid_algorithm::Response &res){
    double P = req.K_p * req.error;
    res.totalFValue = req.error * req.T + req.totalFValue;
    double I = req.K_i * res.totalFValue;
    // split D into top and bottom so its more readable
    double && topD = req.error - req.lastError;
    double D = req.K_d * (topD / req.T);

    res.y = P + I + D;
    // checking the documentation 22.0 is the maximum speed
    res.y = std::min(res.y, 0.22);
    res.y = std::max(res.y, 0.0);

    ROS_INFO("lastError: %u totalFValue %lf",
            (unsigned int) res.lastError, res.totalFValue);
    ROS_INFO("K_p: %lf K_i %lf K_d %lf", req.K_p, req.K_i, req.K_d);
    ROS_INFO("P: %lf, I:, %lf, D: %lf", P,I, D);
    ROS_INFO("Returning y as %lf", res.y);
    return true;
}

int main(int argc, char **argv){
    ros::init(argc, argv, "pid_algorithm_node");
    ros::NodeHandle nodeHandle;

    ros::ServiceServer getSonarReadings =
        nodeHandle.advertiseService("pid_algorithm", pidAlgorithm);

    ROS_INFO("Ready to manage requests");
    ros::spin();

    return EXIT_SUCCESS;
}
```

Figure 2: The PID srv and cpp files respectively

4 Task 2: Kalman Filter Node

The goal of a Kalman filter is to get rid of noise in sonar readings through maths, and in this case odometry readings. The equations to be used are defined in Equations 6.

$$\begin{aligned} K_i &= \frac{P_i^-}{P_i^- + R_i} \\ y_i &= y_i^- + K_i(z_i - y_i^-) \\ P_i &= (1 - K_i)P_i^- \end{aligned} \tag{6}$$

Where z_i is the sonar's reading, y_i^- is the estimation of how far the robot is from the bowl at step i, P_i^- is the predicted variance at i, and R_i is the initial variance read. Since there are no continuous variables in this, there is no need for this to be changed such as the PID. However, each of these should still be properly explained.

The naive way to solve the issue of a normally distributed noisy sonar would simply be to measure the median every time. However, to do that the robot would have to sit there for several seconds, move forwards slightly then sit there and measure the median again. Instead, what the Kalman filter aims to do is take the measuring once, move, (in our case) consider how far it moved through the odometry, apply the Kalman filter to this and get an estimate of how far it is [Siegward et al., 2011]. The reason why the odometry alone would not be enough is visible in how the sonar functions. Consider the case where the bowl lies 13 degrees to the right of the way the robot is facing. If we only use the euclidean distance measured by the odometry system, it would not actually be the distance from the bowl. From all of this, we can see the need for why the Kalman filter is necessary.

For now the focus is on applying the Kalman filter step out of the steps detailed previously which is all in equation 6. In short, the goal of this step is to give readings varying weight depending on the certainty of them. This weight is given by the Kalman gain by taking the predicted variance divided by itself added to the overall variance. Then we can add the difference in the reading and the estimated position to its estimated position to find y_i. To calculate the next predicted variance is typically complicated, but was compressed into multiplying P_i^- by the complement of the Kalman gain $(1 - K_i)$. In practice this just makes later readings have less weight.

In an attempt to understand this particular equation, an interesting proof by induction was found that makes calculating the variance irrelevant. Essentially it's possible to remove P_i, P_i^- and R_i from the equations. First, let's change the notation so that the pattern is more obvious.

$$\begin{aligned} K_i &= \frac{P_i}{P_i + R} \\ P_{i+1} &= (1 - K_i)P_i \\ P_0 &= R \end{aligned}$$

First we define our base case of K_0

$$\begin{aligned} K_0 &= \frac{P_0}{P_0 + R} \\ P_0 &= R \\ K_0 &= \frac{P_0}{2P_0} \\ K_0 &= \frac{1}{2} \end{aligned}$$

Now in induction we just have to prove the $i+1$ case

$$\begin{aligned}
P_{i+1} &= R(1-K_i)\\
K_{i+1} &= \frac{P_{i+1}}{P_{i+1}+R}\\
&= \frac{R(1-K_i)}{R(1-K_i)+R}\\
&= \frac{R(1-K_i)}{R-RK_i+R}\\
&= \frac{R(1-K_i)}{2R-RK_i}\\
&= \frac{R(1-K_i)}{R(2-K_i)}\\
&= \frac{1-K_i}{2-K_i}
\end{aligned}$$

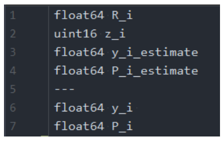
So essentially we found that

$$\begin{aligned}
K_0 &= \frac{1}{2}\\
K_{i+1} &= \frac{1-K_i}{2-K_i}
\end{aligned}$$

To elaborate, the reasons why this is happening is because P_i does not change in the middle between calculations as the odometry has no variance. If it had a changing variance and P_i were to suddenly spike, it would give less weight to the odometry and more weight to the sonar reader. Essentially it's throwing the sonars out the window by decreasing the value of the Kalman gain quickly. However, it still means that the entire variance step could be skipped as a result.

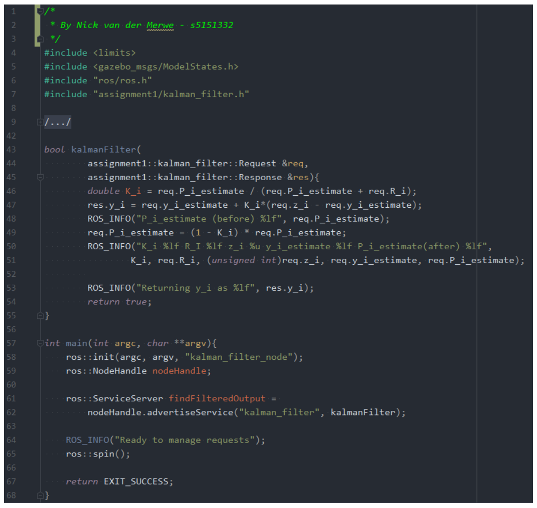
With this being said, it will just be ignored. It's an interesting proof that makes variance useless, but taking it out will probably remove a section of the assignment that the assignment writer intended the student to do. It's essentially just evidence of extensive research around understanding the formulas at work.

Moving onto the implementation, its a service that uses the exact formulas on the task sheet visible in figure 3.



```
float64 R_i
uint16 z_i
float64 y_i_estimate
float64 P_i_estimate
---
float64 y_i
float64 P_i
```

(a)



```
/*
 * By Nick van der Merwe - s5151332
 */
#include <limits>
#include <gazebo_msgs/ModelStates.h>
#include "ros/ros.h"
#include "assignment1/kalman_filter.h"

/.../

bool kalmanFilter(
        assignment1::kalman_filter::Request &req,
        assignment1::kalman_filter::Response &res){
        double K_i = req.P_i_estimate / (req.P_i_estimate + req.R_i);
        res.y_i = req.y_i_estimate + K_i*(req.z_i - req.y_i_estimate);
        ROS_INFO("P_i_estimate (before) %lf", req.P_i_estimate);
        req.P_i_estimate = (1 - K_i) * req.P_i_estimate;
        ROS_INFO("K_i %lf R_I %lf z_i %u y_i_estimate %lf P_i_estimate(after) %lf",
                K_i, req.R_i, (unsigned int)req.z_i, req.y_i_estimate, req.P_i_estimate);

        ROS_INFO("Returning y_i as %lf", res.y_i);
        return true;
}

int main(int argc, char **argv){
    ros::init(argc, argv, "kalman_filter_node");
    ros::NodeHandle nodeHandle;

    ros::ServiceServer findFilteredOutput =
        nodeHandle.advertiseService("kalman_filter", kalmanFilter);

    ROS_INFO("Ready to manage requests");
    ros::spin();

    return EXIT_SUCCESS;
}
```

(b)

Figure 3: Kalman filter implementation, srv file in 3a and code in 3b

5 Controller Node

5.1 PID constants

As picking the PID terms is not an essential step and, only needs to be justified, we can use manual selection. To define our PID terms we must consider the goal of how we wish the robot to move. A reasonably logical approach would be to make the robot run at maximum speed until it comes close to the bowl and then slow down when its closer so that it can accurately move. To align this with a real world goal, it's wanting to drive to a location as fast as possible yet be safe. In other words, when a car is parking it slows down so it can move more accurately.

The K_p term does exactly this by changing its speed in respect to its distance from the target. As such we can pick a point that looks reasonably close (see figure 4) and define that as the point it should start slowing. To work out the K_p from this all that must be done is the maximum speed should be found when its at 100 units by doing $y(t) = K_p e(t) => 0.22 = K_p 100 => K_p = 0.22/100$.

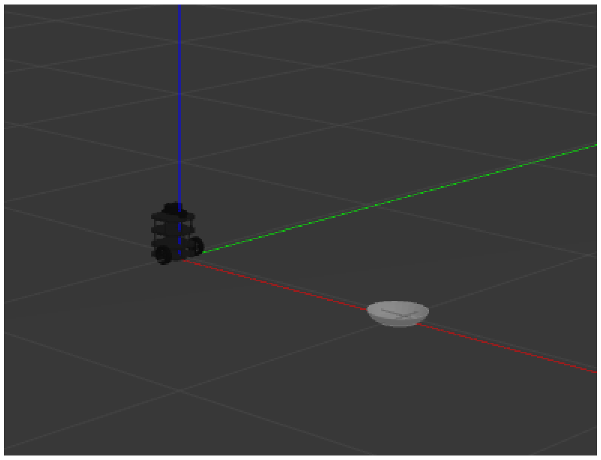


Figure 4: How far the robot looks when it's 100 units away

One issue to consider is that typically the error in PID is not just distance (i.e. its the difference between the goal speed and current speed), which is especially impactful to the integral term. Usually when the error hits zero, the integral value is set as a constant - otherwise it would continue to rise as the controller is running [Siegward et al., 2011]. In this case it only occurs when the robot actually reaches the bowl - which would be the end of the problem. In the case that the robot is as far away as its sonar readers allow (around 60,000), it would likely mean that it will just run at maximum speed all the way to the bowl. This results in three options: 1) Set K_i to close to 0, rendering the entire term next to useless 2) Let the robot just run at maximum speed when its far away 3) Decide on a point when the integral term should stop changing.

To be true to the goal, the best option here would be option 1. Option 2 is in direct conflict with the goal of slowing down when close as it renders it impossible in some situations, and picking an arbitrary point where the integral stops growing would only cause the robot to have a minimum constant speed after a certain point. While K_i could be set close to zero such that it has a 'conceptual' effect, it's easier to just set it to zero as it would affect the speed in the same way in this environment.

As the derivative term allows the robot to continue going a certain speed or slow down less than usual, it could be useful in the Kalman filter. This is because occasionally the error is filtered to be a negative value which in that case the P term completely stops, and often far too early. Furthermore, even with just the P term on its own in a non-noisy sonar it starts going abysmally slow as it approaches it. As a solution, K_d is defined such that the robot an only lose 80% of its speed every second. To do this, the meaning of what the D term should be understood fully. It takes in the current distance, subtracts the last distance and divides by T. The subtraction operation, in theory, should always lead to a negative number as we are approaching the goal and dividing by T (0.01 seconds) says what the velocity would be for a second - as a negative. From this all that must be done to make sure 20% of the previous velocity is added,K_d should equal $-0.2/100$ as to cancel the negative and represent its velocity in metres (that is the divided by 100) every second.

In reality though, due to the factor that the sonar readings are integers this does not function as intended. The maximum speed is 22 centimetres per second, and its reporting that 100 times every second. In other words, every four (or so) readings the difference would be 1cm, and not every reading has a difference of 0.22. As a result, the D term simply adds four times more than the intended value every four iterations. Since this is a sonar based issue, there is not much of a work around other than turning the sonar reading into a float64, ignoring the issue or setting K_d to zero. Out of these, the issue was chosen to be ignored as in the Kalman filter this is not what occurs: the readings are floats.

To tie it together, it was decided that $K_p = 0.22/100$, $K_i = 0$, $K_d = -0.2$. K_p makes the robot slow down as it approaches and K_d prevents the robot from slowing down too abruptly. The issue with K_i is that it makes the robot either stay at a certain speed, or stay at maximum speed depending on how its implemented.

5.2 Calculating Variance

To do this is not any different from usual:

$$\begin{aligned} mean &= \frac{1}{n}\sum_{i=0}^{n} x_i \\ variance &= \frac{\sum_{i=0}^{n}(x_i - mean)^2}{n-1} \end{aligned} \tag{7}$$

where x_i is a collected sample and n is the number of samples [Bhandari, 2020]. Note that the sample formula is used instead of the population formula. With regards to the number of samples collected, it was decided that 1000 will be used. Realistically, this is hard to justify without other variables involved but at 1000 points it is expected to give ten points for every percentile which should be reasonable.

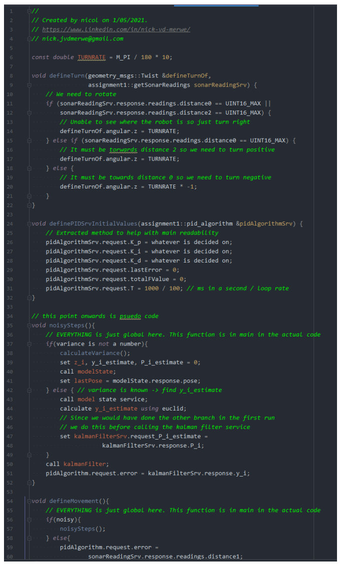
One option is to hard-code in the variance because in the *sonars.cc* source code provided $std :: normal\_distribution()$ is used and the standard deviation is defined as 100. Meaning, the variance is hard-coded to $100^2 = 10,000$. However, hard-coding in the answer would destroy an intended part of the assignment and reduce reusability in the case that the sonar reader's distribution changes. Furthermore, this can be used as justification for why the variance is calculated every run: to improve reusability.

5.3 Random turns due to noise

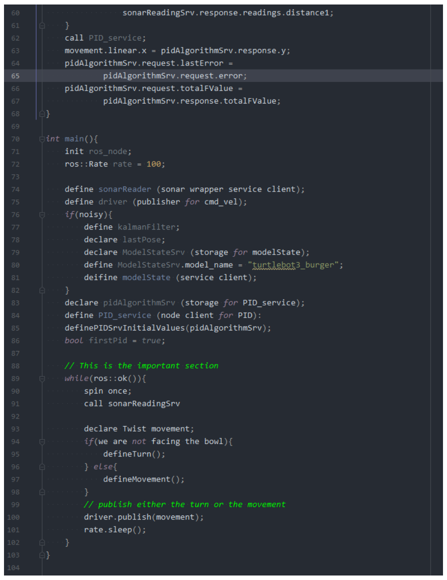
Due to the sonars being capable of randomly shooting to uint16's max (even if the chance is low), it is possible for the robot to accidentally shift out of sight of the bowl if one comes through. To counteract this, it was implemented that the robot should read that it lost sight of the bowl five consecutive times before it turns. While technically possible, the chances of this occurring is low enough that it does not happen.

5.4 Structure

A core factor that makes implementing this difficult is that extracting functions from main in ROS is often prohibited due to a couple of methods only being permitted to be called inside main. So for easier understanding, figure 5 was made to grasp the implementation better. However, it should only be used as a guide to the general structure - some extra things were added in the code that was detailed in the justifications, but not the pseudo-code.



```
//
// Created by nicol on 1/05/2021.
// https://www.linkedin.com/in/nick-vd-merwe/
// nick.jvdmerwe@gmail.com

const double TURNRATE = M_PI / 180 * 10;

void defineTurn(geometry_msgs::Twist &defineTurnOf,
                assignment1::getSonarReadings sonarReadingSrv) {
    // We need to rotate
    if (sonarReadingSrv.response.readings.distance0 == UINT16_MAX ||
        sonarReadingSrv.response.readings.distance2 == UINT16_MAX) {
        // Unable to see where the robot is so just turn right
        defineTurnOf.angular.z = TURNRATE;
    } else if (sonarReadingSrv.response.readings.distance0 == UINT16_MAX) {
        // It must be torwards distance 2 so we need to turn positive
        defineTurnOf.angular.z = TURNRATE;
    } else {
        // It must be towards distance 0 so we need to turn negative
        defineTurnOf.angular.z = TURNRATE * -1;
    }
}

void definePIDSrvInitialValues(assignment1::pid_algorithm &pidAlgorithmSrv) {
    // Extracted method to help with main readability
    pidAlgorithmSrv.request.K_p = whatever is decided on;
    pidAlgorithmSrv.request.K_i = whatever is decided on;
    pidAlgorithmSrv.request.K_d = whatever is decided on;
    pidAlgorithmSrv.request.lastError = 0;
    pidAlgorithmSrv.request.totalFValue = 0;
    pidAlgorithmSrv.request.T = 1000 / 100; // ms in a second / loop rate
}

// this point onwards is psuedo code
void noisySteps(){
    // EVERYTHING is just global here. This function is in main in the actual code
    if(variance is not a number){
        calculateVariance();
        set z_i, y_i_estimate, P_i_estimate = 0;
        call modelState;
        set lastPose = modelState.response.pose;
    } else { // variance is known -> find y_i_estimate
        call model state service;
        calculate y_i_estimate using euclid;
        // Since we would have done the other branch in the first run
        // we do this before calling the kalman filter service
        set kalmanFilterSrv.request_P_i_estimate =
                    kalmanFilterSrv.response.P_i;
    }
    call kalmanFilter;
    pidAlgorithm.request.error = kalmanFilterSrv.response.y_i;
}

void defineMovement(){
    // EVERYTHING is just global here. This function is in main in the actual code
    if(noisy){
        noisySteps();
    } else{
        pidAlgorithm.request.error =
                sonarReadingSrv.response.readings.distance1;
```

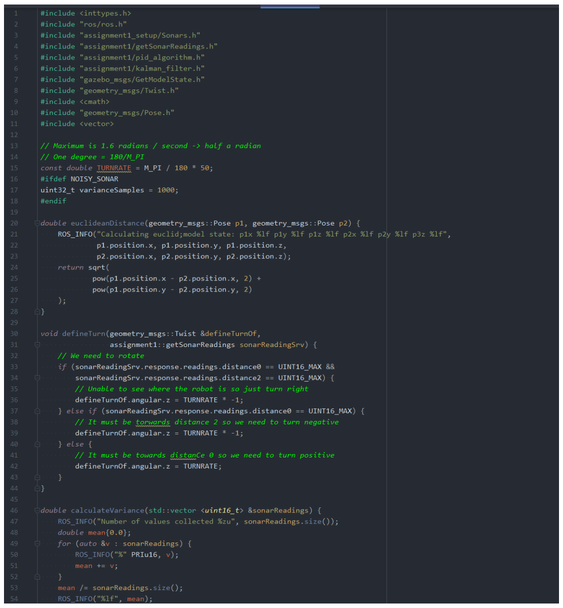


```
                sonarReadingSrv.response.readings.distance1;
    }
    call PID_service;
    movement.linear.x = pidAlgorithmSrv.response.y;
    pidAlgorithmSrv.request.lastError =
            pidAlgorithmSrv.request.error;
    pidAlgorithmSrv.request.totalFValue =
            pidAlgorithmSrv.response.totalFValue;
}

int main(){
    init ros_node;
    ros::Rate rate = 100;

    define sonarReader (sonar wrapper service client);
    define driver (publisher for cmd_vel);
    if(noisy){
        define kalmanFilter;
        declare lastPose;
        declare ModelStateSrv (storage for modelState);
        define ModelStateSrv.model_name = "turtlebot3_burger";
        deifine modelState (service client);
    }
    declare pidAlgorithmSrv (storage for PID_service);
    define PID_service (node client for PID):
    definePIDSrvInitialValues(pidAlgorithmSrv);
    bool firstPid = true;

    // This is the important section
    while(ros::ok()){
        spin once;
        call sonarReadingSrv

        declare Twist movement;
        if(we are not facing the bowl){
            defineTurn();
        } else{
            defineMovement();
        }
        // publish either the turn or the movement
        driver.publish(movement);
        rate.sleep();
    }
}
```

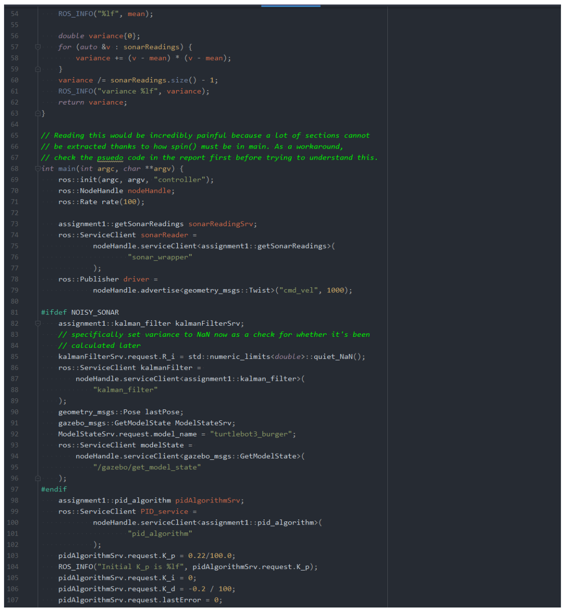
Figure 5: Pseudocode for better understanding of main code

5.5 Implementation

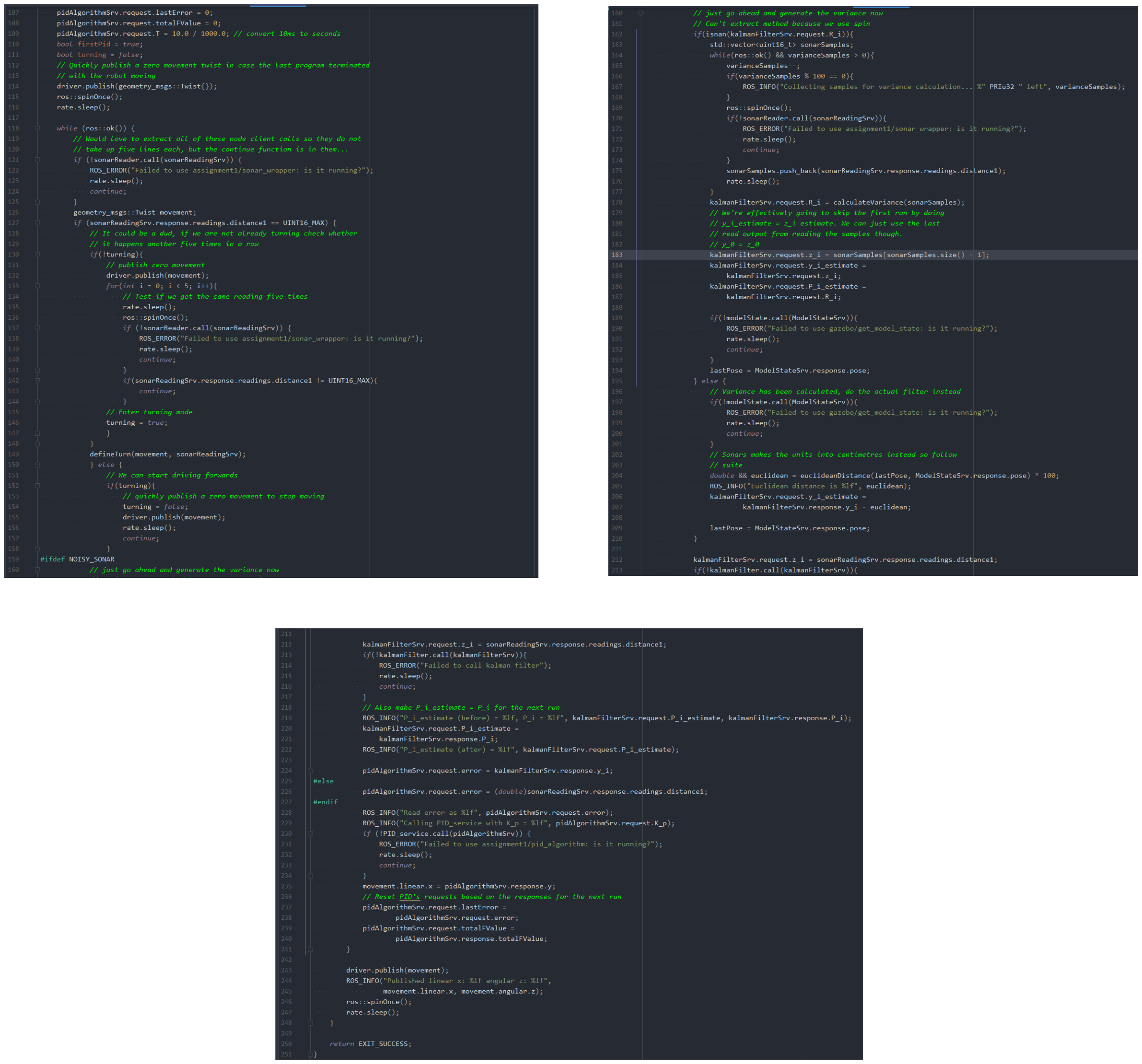
Following the pseudocode, the controller was implemented as visible in figure 5.



```
#include <inttypes.h>
#include "ros/ros.h"
#include "assignment1_setup/Sonars.h"
#include "assignment1/getSonarReadings.h"
#include "assignment1/pid_algorithm.h"
#include "assignment1/kalman_filter.h"
#include "gazebo_msgs/GetModelState.h"
#include "geometry_msgs/Twist.h"
#include <cmath>
#include "geometry_msgs/Pose.h"
#include <vector>

// Maximum is 1.6 radians / second -> half a radian
// One degree = 180/M_PI
const double TURNRATE = M_PI / 180 * 50;
#ifdef NOISY_SONAR
uint32_t varianceSamples = 1000;
#endif

double euclideanDistance(geometry_msgs::Pose p1, geometry_msgs::Pose p2) {
    ROS_INFO("Calculating euclid;model state: p1x %lf p1y %lf p1z %lf p2x %lf p2y %lf p3z %lf",
             p1.position.x, p1.position.y, p1.position.z,
             p2.position.x, p2.position.y, p2.position.z);
    return sqrt(
            pow(p1.position.x - p2.position.x, 2) +
            pow(p1.position.y - p2.position.y, 2)
    );
}

void defineTurn(geometry_msgs::Twist &defineTurnOf,
                assignment1::getSonarReadings sonarReadingSrv) {
    // We need to rotate
    if (sonarReadingSrv.response.readings.distance0 == UINT16_MAX &&
        sonarReadingSrv.response.readings.distance2 == UINT16_MAX) {
        // Unable to see where the robot is so just turn right
        defineTurnOf.angular.z = TURNRATE * -1;
    } else if (sonarReadingSrv.response.readings.distance0 == UINT16_MAX) {
        // It must be torwards distance 2 so we need to turn negative
        defineTurnOf.angular.z = TURNRATE * -1;
    } else {
        // It must be towards distanCe 0 so we need to turn positive
        defineTurnOf.angular.z = TURNRATE;
    }
}

double calculateVariance(std::vector <uint16_t> &sonarReadings) {
    ROS_INFO("Number of values collected %zu", sonarReadings.size());
    double mean{0.0};
    for (auto &v : sonarReadings) {
        ROS_INFO("%" PRIu16, v);
        mean += v;
    }
    mean /= sonarReadings.size();
    ROS_INFO("%lf", mean);
```



```
    ROS_INFO("%lf", mean);

    double variance{0};
    for (auto &v : sonarReadings) {
        variance += (v - mean) * (v - mean);
    }
    variance /= sonarReadings.size() - 1;
    ROS_INFO("variance %lf", variance);
    return variance;
}

// Reading this would be incredibly painful because a lot of sections cannot
// be extracted thanks to how spin() must be in main. As a workaround,
// check the psuedo code in the report first before trying to understand this.
int main(int argc, char **argv) {
    ros::init(argc, argv, "controller");
    ros::NodeHandle nodeHandle;
    ros::Rate rate(100);

    assignment1::getSonarReadings sonarReadingSrv;
    ros::ServiceClient sonarReader =
            nodeHandle.serviceClient<assignment1::getSonarReadings>(
                    "sonar_wrapper"
            );
    ros::Publisher driver =
            nodeHandle.advertise<geometry_msgs::Twist>("cmd_vel", 1000);

#ifdef NOISY_SONAR
    assignment1::kalman_filter kalmanFilterSrv;
    // specifically set variance to NaN now as a check for whether it's been
    // calculated later
    kalmanFilterSrv.request.R_i = std::numeric_limits<double>::quiet_NaN();
    ros::ServiceClient kalmanFilter =
        nodeHandle.serviceClient<assignment1::kalman_filter>(
            "kalman_filter"
    );
    geometry_msgs::Pose lastPose;
    gazebo_msgs::GetModelState ModelStateSrv;
    ModelStateSrv.request.model_name = "turtlebot3_burger";
    ros::ServiceClient modelState =
        nodeHandle.serviceClient<gazebo_msgs::GetModelState>(
            "/gazebo/get_model_state"
    );
#endif
    assignment1::pid_algorithm pidAlgorithmSrv;
    ros::ServiceClient PID_service =
            nodeHandle.serviceClient<assignment1::pid_algorithm>(
                    "pid_algorithm"
            );
    pidAlgorithmSrv.request.K_p = 0.22/100.0;
    ROS_INFO("Initial K_p is %lf", pidAlgorithmSrv.request.K_p);
    pidAlgorithmSrv.request.K_i = 0;
    pidAlgorithmSrv.request.K_d = -0.2 / 100;
    pidAlgorithmSrv.request.lastError = 0;
```



```
    pidAlgorithmSrv.request.lastError = 0;
    pidAlgorithmSrv.request.totalFValue = 0;
    pidAlgorithmSrv.request.T = 10.0 / 1000.0; // convert 10ms to seconds
    bool firstPid = true;
    bool turning = false;
    // Quickly publish a zero movement twist in case the last program terminated
    // with the robot moving
    driver.publish(geometry_msgs::Twist{});
    ros::spinOnce();
    rate.sleep();

    while (ros::ok()) {
        // Would love to extract all of these node client calls so they do not
        // take up five lines each, but the continue function is in them...
        if (!sonarReader.call(sonarReadingSrv)) {
            ROS_ERROR("Failed to use assignment1/sonar_wrapper: is it running?");
            rate.sleep();
            continue;
        }
        geometry_msgs::Twist movement;
        if (sonarReadingSrv.response.readings.distance1 == UINT16_MAX) {
            // It could be a dud, if we are not already turning check whether
            // it happens another five times in a row
            if(!turning){
                // publish zero movement
                driver.publish(movement);
                for(int i = 0; i < 5; i++){
                    // Test if we get the same reading five times
                    rate.sleep();
                    ros::spinOnce();
                    if (!sonarReader.call(sonarReadingSrv)) {
                        ROS_ERROR("Failed to use assignment1/sonar_wrapper: is it running?");
                        rate.sleep();
                        continue;
                    }
                    if(sonarReadingSrv.response.readings.distance1 != UINT16_MAX){
                        continue;
                    }
                // Enter turning mode
                turning = true;
                }
            }
            defineTurn(movement, sonarReadingSrv);
            } else {
                // We can start driving forwards
                if(turning){
                    // quickly publish a zero movement to stop moving
                    turning = false;
                    driver.publish(movement);
                    rate.sleep();
                    continue;
                }
#ifdef NOISY_SONAR
            // just go ahead and generate the variance now
```

```
            // just go ahead and generate the variance now
            // Can't extract method because we use spin
            if(isnan(kalmanFilterSrv.request.R_i)){
                std::vector<uint16_t> sonarSamples;
                while(ros::ok() && varianceSamples > 0){
                    varianceSamples--;
                    if(varianceSamples % 100 == 0){
                        ROS_INFO("Collecting samples for variance calculation... %" PRIu32 " left", varianceSamples);
                    }
                    ros::spinOnce();
                    if(!sonarReader.call(sonarReadingSrv)){
                        ROS_ERROR("Failed to use assignment1/sonar_wrapper: is it running?");
                        rate.sleep();
                        continue;
                    }
                    sonarSamples.push_back(sonarReadingSrv.response.readings.distance1);
                    rate.sleep();
                }
                kalmanFilterSrv.request.R_i = calculateVariance(sonarSamples);
                // We're effectively going to skip the first run by doing
                // y_i_estimate = z_i estimate. We can just use the last
                // read output from reading the samples though.
                // y_0 = z_0
                kalmanFilterSrv.request.z_i = sonarSamples[sonarSamples.size() - 1];
                kalmanFilterSrv.request.y_i_estimate =
                    kalmanFilterSrv.request.z_i;
                kalmanFilterSrv.request.P_i_estimate =
                    kalmanFilterSrv.request.R_i;

                if(!modelState.call(ModelStateSrv)){
                    ROS_ERROR("Failed to use gazebo/get_model_state: is it running?");
                    rate.sleep();
                    continue;
                }
                lastPose = ModelStateSrv.response.pose;
            } else {
                // Variance has been calculated, do the actual filter instead
                if(!modelState.call(ModelStateSrv)){
                    ROS_ERROR("Failed to use gazebo/get_model_state: is it running?");
                    rate.sleep();
                    continue;
                }
                // Sonars makes the units into centimetres instead so follow
                // suite
                double && euclidean = euclideanDistance(lastPose, ModelStateSrv.response.pose) * 100;
                ROS_INFO("Euclidean distance is %lf", euclidean);
                kalmanFilterSrv.request.y_i_estimate =
                        kalmanFilterSrv.response.y_i - euclidean;

                lastPose = ModelStateSrv.response.pose;
            }

            kalmanFilterSrv.request.z_i = sonarReadingSrv.response.readings.distance1;
            if(!kalmanFilter.call(kalmanFilterSrv)){
```

```

            kalmanFilterSrv.request.z_i = sonarReadingSrv.response.readings.distance1;
            if(!kalmanFilter.call(kalmanFilterSrv)){
                ROS_ERROR("Failed to call kalman filter");
                rate.sleep();
                continue;
            }
            // Also make P_i_estimate = P_i for the next run
            ROS_INFO("P_i_estimate (before) = %lf, P_i = %lf", kalmanFilterSrv.request.P_i_estimate, kalmanFilterSrv.response.P_i);
            kalmanFilterSrv.request.P_i_estimate =
                kalmanFilterSrv.response.P_i;
            ROS_INFO("P_i_estimate (after) = %lf", kalmanFilterSrv.request.P_i_estimate);

            pidAlgorithmSrv.request.error = kalmanFilterSrv.response.y_i;
#else
            pidAlgorithmSrv.request.error = (double)sonarReadingSrv.response.readings.distance1;
#endif
            ROS_INFO("Read error as %lf", pidAlgorithmSrv.request.error);
            ROS_INFO("Calling PID_service with K_p = %lf", pidAlgorithmSrv.request.K_p);
            if (!PID_service.call(pidAlgorithmSrv)) {
                ROS_ERROR("Failed to use assignment1/pid_algorithm: is it running?");
                rate.sleep();
                continue;
            }
            movement.linear.x = pidAlgorithmSrv.response.y;
            // Reset PID's requests based on the responses for the next run
            pidAlgorithmSrv.request.lastError =
                    pidAlgorithmSrv.request.error;
            pidAlgorithmSrv.request.totalFValue =
                    pidAlgorithmSrv.response.totalFValue;
        }

        driver.publish(movement);
        ROS_INFO("Published linear x: %lf angular z: %lf",
                 movement.linear.x, movement.angular.z);
        ros::spinOnce();
        rate.sleep();
    }

    return EXIT_SUCCESS;
}
```

Figure 8: Implementation of the controller. Read as top left -¿ top right -¿ bottom left -¿ bottom right

6 Compiling and replicating the results

The package should be extracted into a catkin workspace package and catkin_make should be used. To launch the non-noisy version, launch sonars (from assignment1_setup) assignment1_pidAlgorithm, assignment1_sonarWrapper and assignment1_controller. To do the Kalman filter launch noisy_sonars (from assignment1_setup) assignment1_pidAlgorithm, assignment1_sonarWrapper, assignment1_kalmanFilter, and assignment1_kalman_controller.

References

[Bhandari, 2020] Bhandari, P. (2020). Understanding and calculating variance. https://www.scribbr.com/statistics/variance/.

[M., 2000] M., A. (2000). Pid control. https://www.eolss.net/ebooks/Sample%20Chapters/C18/E6-43-03-03.pdf.

[Siegward et al., 2011] Siegward, Roland, Nourbakhshs, I. R., and Scaramuzza, D. (2011). Introduction to autonomous mobile robots second edition. https://learn-ap-southeast-2-prod-fleet01-xythos.content.blackboardcdn.com/5bb70f08ac35e/8023721?X-Blackboard-Expiration=1619870400000&X-Blackboard-Signature=55OomC3C9LgQebBieXZLrHODLw7EyXD7bBOn8vVfmOU%3D&X-Blackboard-Client-Id=101056&response-cache-control=private%2C%20max-age%3D21600&response-content-disposition=inline%3B%20filename%2A%3DUTF-8%27%27Introduction%2520to%2520Autonomous%2520Mobile%2520Robots_2nd.pdf&response-content-type=application%2Fpdf&X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Date=20210501T060000Z&X-Amz-SignedHeaders=host&X-Amz-Expires=21600&X-Amz-Credential=AKIAYDKQORRYZBCCQFY5%2F20210501%2Fap-southeast-2%2Fs3%2Faws4_request&X-Amz-Signature=a86bfb6cd33b718b655f848704fa7b055a939b711e9b60af19b85768bdc61ec5.